JOHNSON GRANT
IN-61-CR
96743

Advanced Software Development Workstation Engineering Scripting Language Graphical Editor DRAFT Design Document

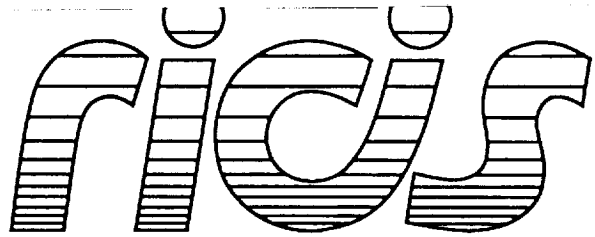
P.56

Inference Corporation

7/9/91

Cooperative Agreement NCC 9-16
Research Activity No. SE.41

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

INTERIM REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Inference Corporation. Dr. Anthony Lekkos, Associate Professor, Computer and Information Sciences, served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Robert Savely of the Information Technology Division, Information Systems Directorate, NASA/JSC.

Advanced Software Development Workstation
Engineering Scripting Language (ESL) Editor
DRAFT Design Document

Prepared for
NASA - Johnson Space Center

9 July 1991

Submitted by
Inference Corporation
550 North Continental Boulevard
El Segundo, CA 90245

Table of Contents

1. Introduction **1**
2. ESL Objects **2**
2.1 Overview 2
2.2 Subprogram Objects 4
2.3 Node Objects 5
2.4 Port Objects 6
2.5 Connector and Connector Group Objects 7
2.6 Implementation Objects 8
2.7 Data Type Objects 9
3. ESL Menus **11**
3.1 Overview 11
3.2 The View Menu 13
3.3 The Edit Menu 14
3.4 The Translate Menu 15
4. Graphical Creation of Objects **16**
5. Description of Panels to Support ESL Graphical Editing **17**
5.1 The Connector Details Panels 17
5.2 Node to Node Connector Details Panel 17
5.3 Constants to Node Connector Details Panel 21
5.4 Graph Input Ports to Node Connector Details Panel 23
5.5 The Node Details Panel 25
5.6 Component Details Panel 28
5.7 Graph Port Details Panel 31
6. Graph Validation and Code Generation **35**
I. ART-IM Schema Representation of ESL Objects **36**

List of Figures

Figure 3-1: ESL Editor Panel 12
Figure 5-1: Node to Node Connector Details Panel 18
Figure 5-2: Constants to Node Connector Details Panel 22
Figure 5-3: Select Constant Value Panel 23
Figure 5-4: Graph Input Ports to Node Connector Details Panel 24
Figure 5-5: Node to Graph Output Ports Connector Details Panel 26
Figure 5-6: Node Details Panel 27
Figure 5-7: Node Notes Panel 29
Figure 5-8: Component Details Panel 30
Figure 5-9: Component Notes Panel 32
Figure 5-10: Graph Ports Details Panel 33

1. Introduction

The Engineering Scripting Language (ESL) is a language designed to allow non-programming users to write Higher Order Language (HOL) programs by drawing directed graphs to represent the program and having the system generate the corresponding program in the HOL. For the implementation of ESL proposed, the HOL code to be generated will be Ada. However, the design of the object system is intended to be as generic as possible and to allow the possibility of generating code in other Higher Order Languages (e.g., C) with minimal modification of the underlying graphical editor.

The building blocks for directed graphs are nodes and connectors. Nodes are visually represented as labeled icons (e.g., rectangles or circles) and have input and output ports which are used to receive and produce data. On a graph, an output port from one node may be connected to an input port on another node via a connector. Visually, all connectors passing data between two nodes are represented as a single arrow connecting the icons representing the nodes. In addition, a graph itself can have input ports and output ports which are connected to ports for nodes on the graph. Visually, the set of all graph input ports is represented by a single icon on the left of the editor window. Each arrow from this icon to a node on the graph represents a group of connectors. Similarly, the set of all graph output ports is represented visually as a single icon on the right of the editor window.

Each node on a graph may represent a primitive procedure or function in the HOL (i.e., a primitive subprogram), an ESL control or data-passing mechanism, or another graph. When a node is a primitive subprogram node, the node's ports represent the subprogram's parameters and, if applicable, its return value.

Chapter 2 of this document contains a description of the objects used by the ESL editor. Chapter 3 presents a description of the functionality supported by the ESL Editor panel. This chapter includes a definition of the functionality supported by pulldown menus on this panel. The following chapter describes how graphical objects will be created. The popup panels designed to support creation of graph objects are described in Chapter 5. Chapter 6 will eventually contain a description of graph semantics and code generation. Finally, Appendix I contains the implementation of the ESL object system in terms of ART-IM schemas.

This document will be revised as design of the ESL editor progresses. It is possible that not all features described in the design will be included in the initial implementation of the software.

2. ESL Objects

2.1 Overview

The ESL system supports user generation of HOL programs through the manipulation of directed graphs. The components of these graphs - the nodes, ports, and connectors - are objects each of which has its own properties and property values. The focus of this chapter is to identify these object types and define their properties.

The purpose of the ESL graphical editor is to allow the user to create or edit graph objects which represent programs. This will be done with the ESL graphical user interface, either by manipulating icons that represent objects or by changing the values in the fields of a popup window.

The basic ESL object types are: the subprogram, the node, the port, the connector, the implementation, and the data type. The node, the port, and the connector objects all map directly to concepts already introduced. A subprogram object represents a subprogram in the Higher Order Language - either a primitive procedure or function or a graph built up from other subprograms. An implementation object contains information about how a subprogram is implemented in the HOL. The data type object contains information about a data type in the HOL.

Node Objects

There are several classes of node objects: the subprogram-node (which includes procedure-node, function-node, and subgraph-node objects), the Merge node, the Replicator node, and the control nodes (If, Select, and Iterator).

A subprogram-node object is used to represent a procedure or function coded in the HOL or to represent a graph previously created through the ESL editor. Each subprogram-node object points to a subprogram object. Subprogram objects are objects visible through the ACCESS Tools Panel and included in the ACCESS taxonomy.

Subprogram objects have corresponding ports. Ports of a procedure or a function object represent parameters of the corresponding procedure or function or the return value of the function. Ports of a graph object, called graph ports, are mapped to ports on nodes of the graph by connector objects.

Merge nodes and Replicator nodes are "glue" nodes that provide for flexibility in connecting nodes. A Merge node has one or more input ports and one output port; a data object on any of the input ports is immediately passed to the output port. A Replicator node has one input port and several output ports; a data object on the input port is immediately passed to each of the output ports. Most Merge and Replicator

nodes will be defined implicitly, by having two or more connectors with the same destination port (Merge) and two or more connectors with the same source port (Replicator).

Finally, there are the control nodes - If, Select, and Iterator. The If and Iterator nodes represent an "if" construct in the HOL and the select node represents a "case" or "switch" statement.

Implementation Objects

An implementation object contains information about how a subprogram object is implemented. The Merge, Replicator, If, Select and Iterator nodes each have an implicit implementation and do not have an associated implementation object. There are three classes of implementation objects: inline, separately compiled procedure, and package.

Inline implementation objects are appropriate only for graph objects. This type of implementation means that when a subgraph node is part of a larger graph for which code is generated, the code corresponding to the subgraph node is generated inline.

Implementation objects whose type is separately compiled procedure are valid for all subprogram objects. Such an implementation object indicates that the subprogram is implemented as a separately compiled Ada procedure. For a separately compiled procedure to be called by an Ada program, the program must first "with" the procedure; then the procedure may be called.

Package implementation objects are valid for subprograms of procedure or function type. Such an implementation object indicates that the subprogram has been implemented as a visible function in an Ada package. For a procedure or function in a package to be called by an Ada program, the program must first "with" the package; then the procedure may be called using the "package.procedure" notation.

Object Hierarchy

This is the hierarchy of objects in the ESL system:

```
subprogram
        primitive subprogram
                function
                procedure
        graph
node
        subprogram node
                primitive subprogram node
```

```
                        procedure node
                        function node
                subgraph node
        merge node
        replicator node
        control node
                if node
                select node
                iterator node
port
        graph port
        procedure port
        function port
        node port
connector group
connector
implementation
        inline implementation
        separately compiled procedure implementation
        package implementation
data type
```

An object's properties define what data is stored with the object. In the ESL system, objects will represented by ART-IM schemas and, possibly, by C++ classes. The purpose of the following sections is to describe the properties of the ESL objects introduced above.

The representation of ESL objects as ART-IM schemas is given in Appendix I.

2.2 Subprogram Objects

This section describes the properties of subprogram objects - e.g., objects which represent a subprogram or program component in some higher order language.

Subprogram properties:

- Subprogram type (e.g., procedure or function, graph).
- Name. In the case of a procedure or function object, this represents the name of the actual procedure or function in the HOL.
- Notes. Value of this property is a documentation string.
- Input ports. Values are objects representing data passed to this subprogram.

- Output ports. Values are objects representing data returned by this subprogram.

- Implementation. Value is an object specifying details on where the implementation for this subprogram can be found (e.g., source/object files).

Additional optional property when the subprogram is a procedure or function:

- Visibility requirements, e.g.,
    - Withed objects (Ada)
    - Required include files (C)

Additional properties when the subprogram object is also a graph object:

- Has nodes. The values for this property are the node objects which are used to construct the graph.

- Has connector groups. The values for this property are connector group objects which represent a group of connectors between a pair of nodes on the graph.

2.3 Node Objects

This section describes the properties of node objects - that is, those objects represented graphically by a rectangle or circle. A graph represents a program or subprogram. Node objects represent either subprograms or represent control or data passing mechanisms for the graph.

Node properties

- Type of node. Different types of nodes are represented as subclasses of the node object class. The different node types are described in Section 2.1.

- Name. The name is user settable.

- Notes. The value of the Notes property is a user-supplied documentation string.

- On graph. The value of this property is the object representing the graph to which this node belongs.

Additional property when the node is a subprogram node:

- Uses subprogram. The value of this property is a subprogram object.

Special Node Objects

As described above, there are special types of nodes which are used for program control and data passing. Additional properties and characteristics of these nodes are described below:

- A Merge node is a node that has an arbitrary number of input ports and one output port. A data object on any of the input ports is immediately produced on the output port. The data type of the ports must be the same.

- A Replicator node is a node that has one input port and an arbitrary number of output ports. A data object on the input port is immediately produced on each of the output ports. The data type of the ports must be the same.

- An If node performs the same function as an "IF" statement in Ada or an "if" statment in C. An If node must contain exactly one input port with data type BOOLEAN; this represents the condition to be tested. It must also have two output ports of trigger data type; one is the "then" port and the other is the "else" port.

- An Iterator node performs much the same function as an If node, but is used to identify parts of the graph where looping back is intended.

- A Select node performs the same function as a "CASE" statement in Ada or a "switch" statement in C. The Select node must contain exactly one input port which corresponds to an enumerated data type. The Select node also contains an arbitrary number of output ports, each of trigger data type. Each output port must correspond to an allowable value for the enumerated data type or to the special value "others" ("default" in C). The port name is the same as the identifier for the value on which the port is triggered. The special port name "others" is used for the default trigger output.

2.4 Port Objects

Each subprogram object points to zero or more input port objects and zero or more output port objects. The ports for a subprogram node are the same as the ports for the corresponding subprogram object. The special control nodes and data-passing nodes do not share port objects with other node objects.

Port properties:

- Name. In the case of a primitive subprogram port, this is identical to the name of the procedure or function parameter to which this port corresponds.

- Direction (input or output).

- Data type. This is an object describing the data type in the HOL.

Additional property when the port is a graph port:

- On graph. Value is the graph object for which this object is a port.

Additional properties when the port is a port for a primitive subprogram object (procedure or function).

- Belongs to subprogram. Value is the corresponding primitive subprogram object.

- Position. Position of the corresponding parameter in the subprogram calling sequence. If the port represents the return value from a function, the value of the position property is 0.

- Parameter type. Value can be IN, OUT, IN-OUT, or (in the case of a port for a function object) RETURN-VALUE. When the parameter type is IN-OUT, there is both an input port object and an output port object corresponding to the same subprogram parameter.

Additional property when the port belongs to a control or data passing node on a graph:

- On node. Value is the node object with which this port is associated.

2.5 Connector and Connector Group Objects

A connector object represents a connection between ports on a graph. Visually, multiple data connections between two nodes on a graph will be represented by a single line. Hence each connector object is associated with a connector group object.

Properties of connector group objects:

- On graph. Value is graph subprogram object.

- Source node. Value is node object on the graph, CONSTANT-VALUE (indicating that the connector group is used to specify constant inputs to a set of node ports) or GRAPH-INPUTS (indicating that the connector group is used to specify a set of graph input ports).

- Destination node. Value is node object on the graph or GRAPH-OUTPUTS (indicating that the connector group is used to specify a set of graph output ports).

- Has connectors. Values are connector objects within this group.

A connector represents a connection between two ports on a graph. These may be ports for graph nodes or may be graph ports. The source node and destination node corresponding to a connector are specified through the connector group object to which the connector object belongs.

Properties of connector objects:

- Notes. Value is a user-supplied documentation string for this connector.

- Belongs to connector group. Value is a connector group object to which this object belongs.

- Source port. This must be a port corresponding to the source node for the connector group, a constant object (when the source node property for the connector group object has value CONSTANT-VALUES) , or a graph port (when the source node property for the connector group object has value GRAPH-INPUTS).

- Destination port. This must be a port corresponding to the destination node for the connector group or a graph port (when the destination node property for the connector group has value GRAPH-OUTPUTS).

2.6 Implementation Objects

As indicated in Section 2.1, an implementation object is used to specify how and where a subprogram is implemented. There are three subclasses of implementation objects: inline, separately compiled procedure, and package. Thus the type of an implementation object is indicated by the subclass to which it belongs.

Additional property when the implementation object is of type inline:

- Code template. When a node's implementation is inline, the HOL code for that node is generated inline with the code for the containing graph. An inline implementation object has a template property whose value contains the information necessary to produce the HOL code.

Additional properties when the implementation object is of type separately compiled procedure:

- Source file name.

- Object file name or library file name.

When a node's implementation is an object of type separately compiled procedure, then the HOL code for that node is a standalone Ada procedure. The source file name property contains the file name for the HOL source for the procedure; it is optional for procedure or function subprogram objects. Likewise the value of the object filename property is the name of the object file for the subprogram.

Additional properties when the implementation object is of type package:

- Name of package.
- Has procedures. Values are primitive subprogram objects which correspond to procedures or functions which are part of this package.
- Package specification file name.
- Package body file name.
- Package specification object file name.
- Package body object file name.

When a node's implementation is package, the HOL code for that node is a visible procedure in an Ada package. Only primitive subprogram objects can have package implementation objects. A package implementation object has properties that specify the package name, the visible procedures in the package, and the source and object filenames for both the package specification and body.

2.7 Data Type Objects

One of the properties of port objects is their data type in the HOL. A data type object must be supplied for each data type which is a property value.

The properties of a data type object are as follows:

- **Name.** The value of this property is a string which is the name of the data type in the HOL (e.g., "INTEGER" in Ada, "int" in C).
- **Defined Values.** The values of this property, if they exist, are strings which are allowable identifiers for this data type.
- **Test Function.** The value of this property, if it exists, is a function which accepts as input a string representing an identifier in the HOL and returns a boolean value TRUE or FALSE, depending on whether the string is or is not a valid identifier for the corresponding data type in the HOL.

When a string is tested to see if it is a valid identifier for a particular data type, it is first compared with the defined values for that data type. If it is on the list of defined values, it is a valid identifier. If it is not on the list of the defined values, then the test function, if it exists, is applied to the string. If the test function returns TRUE, then the string is a valid identifier. If no function is specified or the function specified returns false, then the string is not a valid identifier.

For the ESL Editor, system defined objects will be provided for the standard HOL data types - e.g., INTEGER, FLOAT, BOOLEAN, STRING in Ada. These system definitions may be extended with data supplied by the Knowledge Engineer (e.g., with additional defined values). Other data type objects must be provided as required by the Knowledge Engineer who creates the knowledge base of primitive subprogram objects.

3. ESL Menus

3.1 Overview

The ACCESS user will develop and modify ESL graphs by directly manipulating ESL graph elements on a special ESL Editor Panel. This panel is shown in Figure 3-1. This panel will contain a single Graph View that will display an ESL graph. In this Graph View, the nodes will be represented by icons surrounding a text label, and the connector groups will be represented by arrows that connect two node icons. The ESL Editor Panel will allow the user to create, destroy, move, and connect nodes. The Panel will also contain a mechanism to allow the user to pan the Graph View.

The ESL Editor Panel will contain pulldown menus to support manipulation of graphs. Many of these commands act on specific nodes or connectors. The user indicates which objects are to be acted upon by "selecting" the objects to be acted upon first, then invoking the menu function. The mechanism used to select objects will depend on the the graph tool selected to support implementation, but will probably be similar to that used by other graphical object editors. In these editors, individual objects are selected by clicking the mouse on the object. Additional objects are selected by holding the shift key down while clicking the mouse on additional objects. The "shift-click" mechanism can also be used to unselect previously selected objects. Clicking on empty workspace causes all selected objects to become unselected.

In some contexts, certain pulldown menu commands will be invalid. Depending on the implementation, this will be represented either by dimming the pulldown menu item or by popping up a warning panel.

Underlying the node icons and arrows are ACCESS objects. Manipulating the ESL graph elements on the ESL Editor Panel changes the corresponding ACCESS object immediately. There is no analogy to the "Ok/Apply/Close" mechanism used to modify non-graphical ACCESS objects.

The look and feel of the ESL Editor Panel will depend on the tool selected to support graphical object editing. Currently it is expected that Unidraw will be selected as this tool. Unidraw is a layer of software on top of the InterViews toolkit intended to assist in the construction of graphical object editors.

The ESL Editor Panel will contain the following pulldown menus: View, Edit, and Translate.

The View Menu contains commands that control the Directed Graph View. Functions available from the View Menu include Open, Parent Graph, Subgraph, Print..., and Close.

Figure 3-1

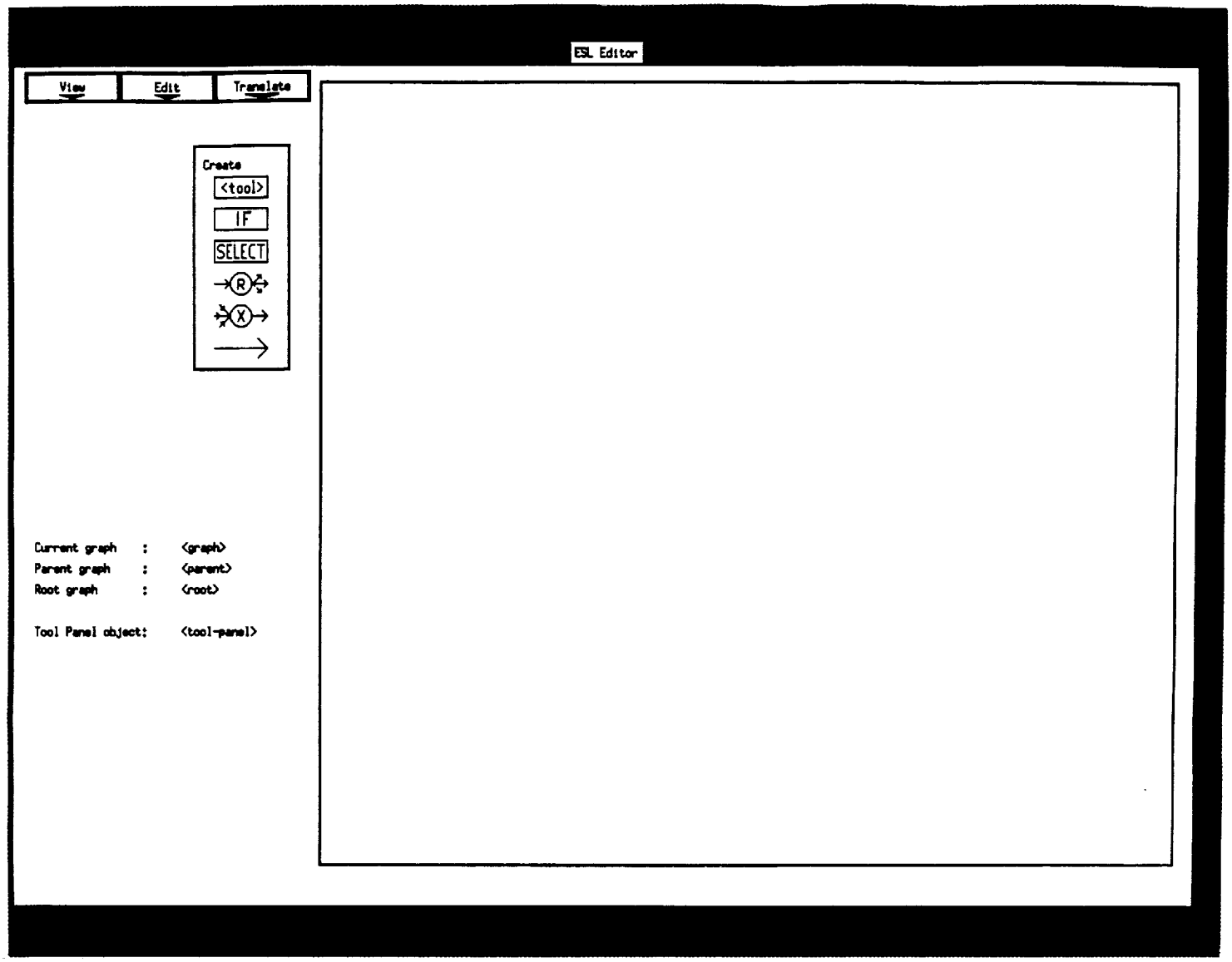

ESL Editor
View
Edit
Translate
Create
<tool>
IF
SELECT
R
X
Current graph : <graph>
Parent graph : <parent>
Root graph : <root>
Tool Panel object: <tool-panel>

Figure 3-1: ESL Editor Panel

The Edit Menu contains commands that allow for the manipulation of a selected object or the entire graph; functions available from the Edit menu include Delete, Copy Graph, Notes..., and Object Details.

The Translate Menu contains commands associated with generating the HOL code: Validate Current Graph, Validate Entire Graph, and Generate Ada.

In addition to the functionality provided through pulldown menus, functionality to create and delete nodes on the graph and to create and delete connectors between these nodes is provided through a palette of icons and tools which is displayed on the left hand side of the ESL Editor Panel. Additional popup panels support this functionality.

Also on the left hand side of the ESL Editor Panel are text items which indicate:

- **Current graph.** That is, the graph object which is currently displayed and being edited.

- **Parent graph.** The parent graph to the current graph, if the current graph was chosen by invoking the "Subgraph" item on the View Menu.

- **Root graph.** The most distant ancestor of the current graph, if the current graph was chosen by invoking the "Subgraph" and "Parent Graph" options from the View Menu.

- **Tools Panel Object.** The object currently selected on the ACCESS Tools Panel.

The following sections describe the functionality of the various menus and menu items in detail.

3.2 The View Menu

The View Menu contains commands that control the Directed Graph View. Functions on this menu are: Open, Parent Graph, Subgraph, Print..., and Close.

- **Open** makes the Directed Graph View display the graph associated with the ACCESS object currently selected on the Tools Panel. The name of this object is displayed at the bottom left of the ESL Editor Panel. If the object selected is not a graph object, this menu item is invalid. When a graph object is first opened, it becomes the Root graph.

- The **Parent Graph** function shifts the Directed Graph View to the parent graph of the current graph. This function is invalid if the graph currently being edited was not chosen by invoking the "Subgraph" item on this menu.

- The **Subgraph** function shifts the Directed Graph View to the graph corresponding of the currently selected subgraph node. This command is invalid if the currently selected node is not a subgraph node, or if more than one node is selected.

- **Print...** allows a user to print a graphical representation of an ESL graph. A dialog box will be popped up to allow the user to select the printing options. An option will be provided to allow a user to print out all the subgraphs under the current graph or just the current graph.

- **Close** dismisses the Directed Graph View.

3.3 The Edit Menu

The Edit Menu contains commands for editing graphs. These are Delete, Copy Graph, Notes..., and Object Details.

- **Delete** is used to delete the selected objects on the graph. If a node is deleted, all connectors to ports on that node are also deleted. If a connector group is deleted, all connectors in that group are deleted.

- **Copy Graph** will create a copy of a graph and all its nodes, input and output ports, and connectors.

- **Notes...** will cause a pop-up panel to be displayed in which the user can enter comments about the graph currently being edited.

- **Object Details...** pops up a window that contains details about the selected node, graph input ports, graph output ports, or connector group. This option is invalid if there is no object selected or if there are more than one object currently selected. The characteristics of the popup window depends on the type of object selected. This is the mechanism that will be used to view properties that are not shown graphically on the ESL graph and to add notes to an object.

The following are potential additions to this menu:

- **Undo** undoes the previous previous editing or creation command.

- **Duplicate** makes a copy of selected nodes and connector groups which connect these nodes.

- **Cut**, **Copy**, and **Paste**. These functions perform clipboard operations. Cut will delete the selected items (nodes and connectors) and copy them into the

clipboard. Copy will copy the selected items into the clipboard without deleting them. Paste will place whatever is in the clipboard into the view. If no item is selected, Cut and Copy are dimmed. If there is no item in the clipboard, Paste is invalid.

In the initial implementation, only Cut (Delete) will be implemented.

3.4 The Translate Menu

This menu contains the functions associated with generating the HOL code. These functions are: Validate Current Graph, Validate Entire Graph, and Generate Ada Code.

- **Validate Current Graph.** This function checks the validity of the current graph at top level. A graph is valid if a syntactically correct HOL program can be generated from the graph and various semantic constraints are satisfied. What constitutes valid graph semantics will be described in more detail in Chapter 6. Validating the current graph does not include validation of subgraph nodes, except to insure that all subgraph input ports are supplied with data of the correct type and that connectors from the subgraph node output ports connect to other ports of the same type.

- **Validate Entire Graph** This function checks the validity of the current graph and, recursively, of all its subgraph nodes.

- **Generate Ada Code** This function is used to generate Ada code for the current graph. This command automatically invokes the Validate Entire Graph function. If the graph is not valid, code generation does not take place.

4. Graphical Creation of Objects

Two methods are currently under evaluation for creating node and connector group objects as displayed in the ESL Editor. The first method is the pulldown menu mechanism. The second method, and that which is currently favored, for graphical creation of objects is the mechanism supported through the Unidraw graphical editing package. With this mechanism, the user is presented with a palette of tools, each represented by an icon. Clicking the select button while over a tool selects it as the currently wielded tool. Clicking the select button in the drawing area invokes the command associated with the tool.

The menu items or tools that create nodes are: Subprogram Node, Merge, Replicator, If, Select, and Iterator. If the menu mechanism is used, these commands will create a node in the center of the screen after which the user can move it. If the tool mechanism is used, the node will be created at the place on the screen where the tool is wielded. The Subprogram node created will be one corresponding to currently selected object on the ACCESS tools panel. The name of this object is displayed on the bottom left of the ESL Editor Panel. If the object selected is not a Subprogram object, the Subprogram Node option will be invalid.

Creation of connector group objects will be handled differently. Both the menu item approach and the tool approach involve introducing a "connect mode". Once in connect mode, the user must specify the source for the connector group and the destination. This may be done by mouse clicking or by some other method.

On the left of the graph display area is an icon which represents the input ports for the current graph; to the right is an icon representing the graph's output ports. When creating a connector group object, the user can select the input ports icon as the source or the output ports icon as a destination. This is the means of creating graph ports.

Any graph node can be a source or destination node for a connector group. A means will also be defined for creating a connector group object which connects constant objects and input ports for a node.

After a connector group object has been created, the user may select it and then select the "Object Details..." option from the Edit Menu, which will cause a Connector Details Panel to be displayed. The user can then create individual connectors using this panel. This panel will be displayed automatically when the connector group object is initially created.

5. Description of Panels to Support ESL Graphical Editing

5.1 The Connector Details Panels

The Connector Details Panels are used to create, modify, or examine the connector groups on a graph. All connectors in a connector group have the same Source and Destination, and all connectors that have the same Source and Destination are in the same group. The Source of a connector can either be a Node, a Constant value, or a Graph Input Port. The Destination of a connector can either be a Node or a Graph Output Port. Depending on the Source and Destination of the connector group, one of the following four Connector Details Panels will be popped up:

- Node to Node Connector Details
- Constant to Node Connector Details
- Graph Input Port to Node Connector Details
- Node to Graph Output Port Connector Details

Since the Node to Node Connector Details is the most general of the Connector Details Panels, it will be discussed first. Then the others will be discussed in turn. The four Connector Details Panels are shown in Figures 5-1, 5-2, 5-4, and 5-5.

5.2 Node to Node Connector Details Panel

The Node to Node Connector Details Panel is used when both the Source and Destination of the connector group are nodes. The Node to Node Connector Details Panel is shown in Figure 5-1. The contents of fields on the panel and responses to events are described below:

- **On Graph.** The On Graph Field contains the name of the graph that the connector is on. The text in this field is read only.
- **Connectors.** The Connectors Field is a textlist field which lists all the connections between the Source and Destination. The title of the textlist identifies the Source and Destination of the connector group. The contents of the textlist are items with fields for the source port name and data type, and the destination port name and data type. The source and destination ports always have the same data type. If there are no connectors between the Source and Destination, then the textlist contains the single item "<none>", which, if selected, becomes unselected immediately.

Figure 5-1

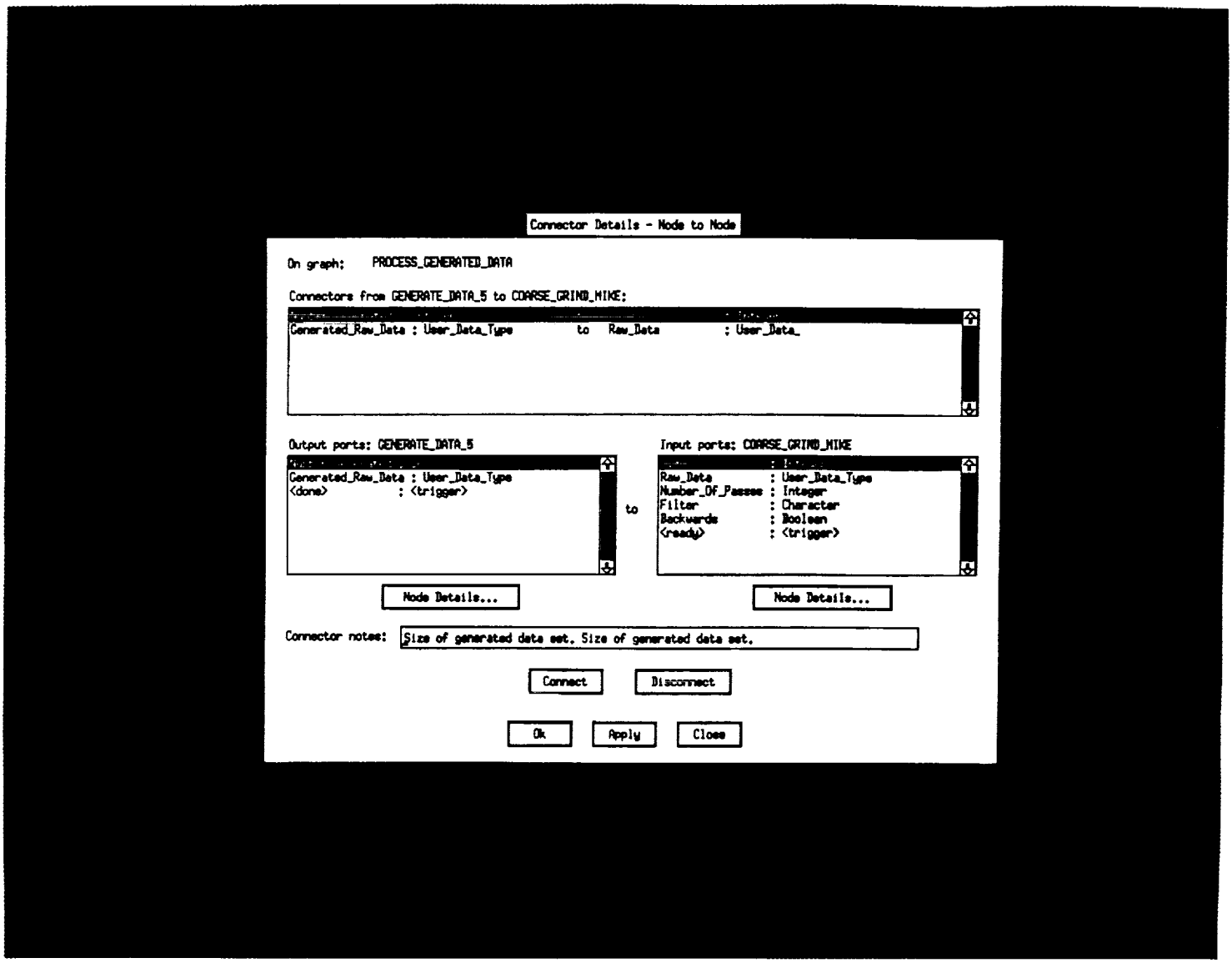

Connector Details - Node to Node
On graph: PROCESS_GENERATED_DATA
Connectors from GENERATE_DATA_5 to COARSE_GRIND_MIKE:
Generated_Raw_Data : User_Data_Type to Raw_Data : User_Data_
Output ports: GENERATE_DATA_5
Generated_Raw_Data : User_Data_Type
<done> : <trigger>
to
Input ports: COARSE_GRIND_MIKE
Raw_Data : User_Data_Type
Number_Of_Passes : Integer
Filter : Character
Backwards : Boolean
<ready> : <trigger>
Node Details...
Node Details...
Connector notes: Size of generated data set. Size of generated data set.
Connect
Disconnect
Ok
Apply
Close

Figure 5-1: Node to Node Connector Details Panel

Selecting an item on the Connectors textlist causes the selection of the items in the Sources textlist and Destinations textlist that correspond to the source and destination of the selected connector item. The Connector Notes text field will be populated with user-entered notes about this connector.

The current selection on the Connectors textlist is read when the Disconnect Button is selected. The Disconnect Button is described below.

- **Source Ports and Destination Ports.** The Source Ports Field is a textlist field listing the output ports of the Source node. Its title identifies the Source node. Similarly, the Destination Ports Field lists the input ports of the Destination node and its title identifies the Destination node. Both fields are populated with items that include the port name and data type.

  When an item on the Source Ports textlist or Destinations Ports textlist is selected, one of two things will happen. If the item selected corresponds to a port that is already the source or destination of a connector between the two nodes, then the corresponding connector item in the Connectors textlist is selected, as is the corresponding source or destination port item on the other textlist. The Connector Notes text field is populated with any user-entered notes about the connector.

  If the item selected does not correspond to a port that is a source or destination of a connector between the two nodes, then any selected item in the Connections textlist is unselected and any corresponding text in the Connector Notes Field is also cleared. If there is an item selected in the other (Source Ports or Destination Ports) textlist and it is not of the same data type as the item just selected, then it is unselected as well.

  All nodes have an output port with trigger data type named "<done>". It is triggered after the node has executed. Likewise, all nodes have an input port with trigger data type named "<ready>". When HOL code is generated from a graph, it will be generated in such a way that the code corresponding to a node with a trigger connection to second node is executed prior to the code corresponding to the second node. This is described in more detail in Chapter 6.

  The current selections on the Source Ports textlist and Destination Ports textlist are read when the Connect button is selected.

- **Node Details...** Below the Source Ports textlist is a button labeled Node Details... When it is selected, a Node Details Panel with information about the Source node will be popped up. A similar button appears below Destination Ports textlist. The Node Details Panel is discussed in Section 5.5.

- **Connector Notes.** The Connector Notes Field is a text field containing one line of user-editable notes about a specific connection. When an item is selected on the Connections textlist, the Connector Notes Field is populated with the notes for that connector. If an item is selected in the Connectors textlist, the Connector Notes Field is read when the user terminates input to that field.

  If no item is selected in the Connectors textlist, then the Connection Notes field can be used to enter notes about a new connector. When this is the case, the Connector Notes field is read when the Connect button selected.

- **Connect.** The Connect Button is used to define a connection between a port on the Source node and a port on the Destination node. When selected, a new connection is defined between the port corresponding to the currently selected item on the Source Ports textlist and the port corresponding to the currently selected item on the Destination Ports textlist. An item for the new connector object is added to the Connectors textlist and is selected. The Connector Notes field is read and its contents are now associated with the new connection definition. The definition is not transformed into an actual connector object until the Apply or Ok Button is selected. A definition which has not been applied is discarded if the panel is dismissed with the Close Button.

- **Disconnect Button.** The Disconnect Button is used to specify deletion of an existing connector object. When selected, the currently selected item on the Connections textlist is deleted and the Connectors textlist is left with no item selected. Any selected items on the Source Ports textlist or on the Destination Ports textlist are unselected. The Connector Notes field is cleared. The corresponding connector object is not permanently deleted until the Apply or Ok Button is selected.

  If there are no more connectors between the Source and Destination, then the Connectors textlist is populated with the single item, "<none>".

- **Apply.** When the Apply Button is selected, any connection definitions or deletions entered through the Connector Details Panel are transformed into actual connector objects or deletions of connector objects. If any constraint violations have occurred, a warning panel is displayed.

- **Close.** When selected, the Close button dismisses the panel without propagating any changes entered since the last apply.

- **Ok.** Selecting the Ok button is equivalent to selecting Apply followed by Close.

5.3 Constants to Node Connector Details Panel

The Constants to Node Connector Details Panel is used when the source ports for a connector group are all constants. The fields on this panel are the same as those on the Node to Node Connector Details Panel, with one exception. Instead of a Sources textlist, a text field is provided to allow the user to enter a constant value. Below this Constant Value Field is a button which when selected will cause a panel to pop up to help the user choose a constant value. The Constant to Node Connector Details Panel is shown in Figure 5-2. The differences in behavior between this panel and the Node to Node Connector Details Panel are listed below:

- **Connectors.** Items in the Connectors field do not contain explicit information about the data type of the connector's source. This allows more room for the constant value. When an item is selected, the Constant Value Field is populated with the value of the Source Port of the corresponding connector.

- **Constant Value.** The Constant Value Field is a text field which is used to enter a constant value to be used as the source of a connector. If the value of this field is changed while an item on the Connectors textlist is selected, its value is tested for validity. This means that it is tested to see if it is a valid identifier for the data type of the input port to which it is intended to be connected.

  A description of how identifiers are tested for validity is contained in Section 2.7.

- **Select...** The Select... Button is used to help a user select a constant value to be applied to a given port. To be used, an item must be selected on the Destination Ports textlist, and a set of constant values for that data type must be defined. If this is the case, then the Select Constant Value Panel is popped up. This panel is shown in Figure 5-3.

  The Select Constant Value Panel has a static text field that indicates the data type of the destination, a textlist that contains the defined constant values for that type, an Ok button, and a Cancel button. When an item is selected on the textlist and the Ok button is selected, the panel is dismissed and the Constant Value Field is populated with the selected value. The Cancel button dismisses the Select Constant Value Panel and leaves the Constant Value Field unchanged.

- **Connect.** Before defining a connection, the constant source value is checked to see if it is a valid identifier for the data type of the destination port. If it is not, a warning panel is displayed and the connection is not defined.

Figure 5-2

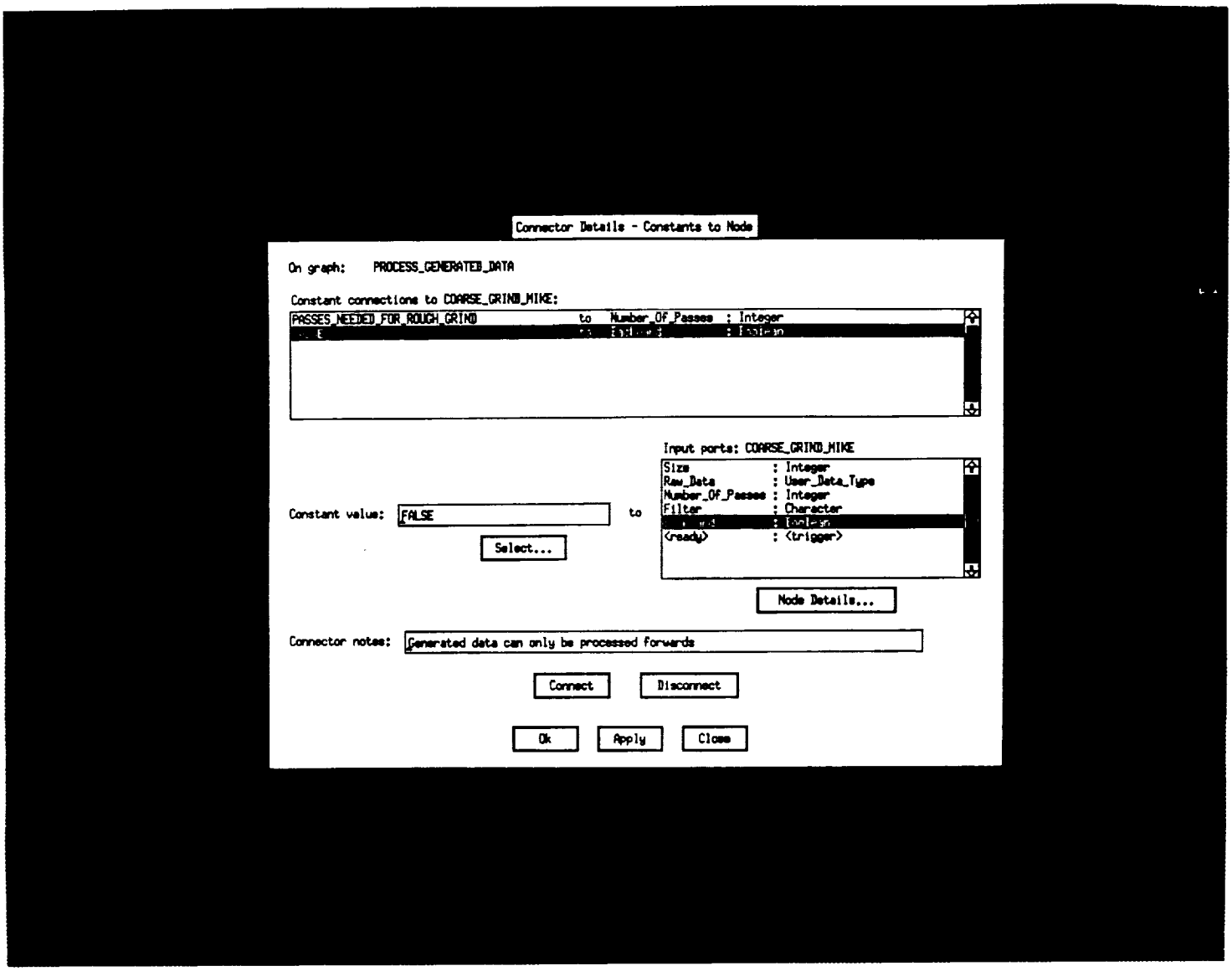

Connector Details - Constants to Node
On graph: PROCESS_GENERATED_DATA
Constant connections to COARSE_GRIND_MIKE:
PASSES_NEEDED_FOR_ROUGH_GRIND to Number_Of_Passes : Integer
Input ports: COARSE_GRIND_MIKE
Size : Integer
Raw_Data : User_Data_Type
Number_Of_Passes : Integer
Filter : Character
<ready> : <trigger>
Constant value: FALSE
to
Select...
Node Details...
Connector notes: Generated data can only be processed forwards
Connect
Disconnect
Ok
Apply
Close

Figure 5-2: Constants to Node Connector Details Panel

Figure 5-3

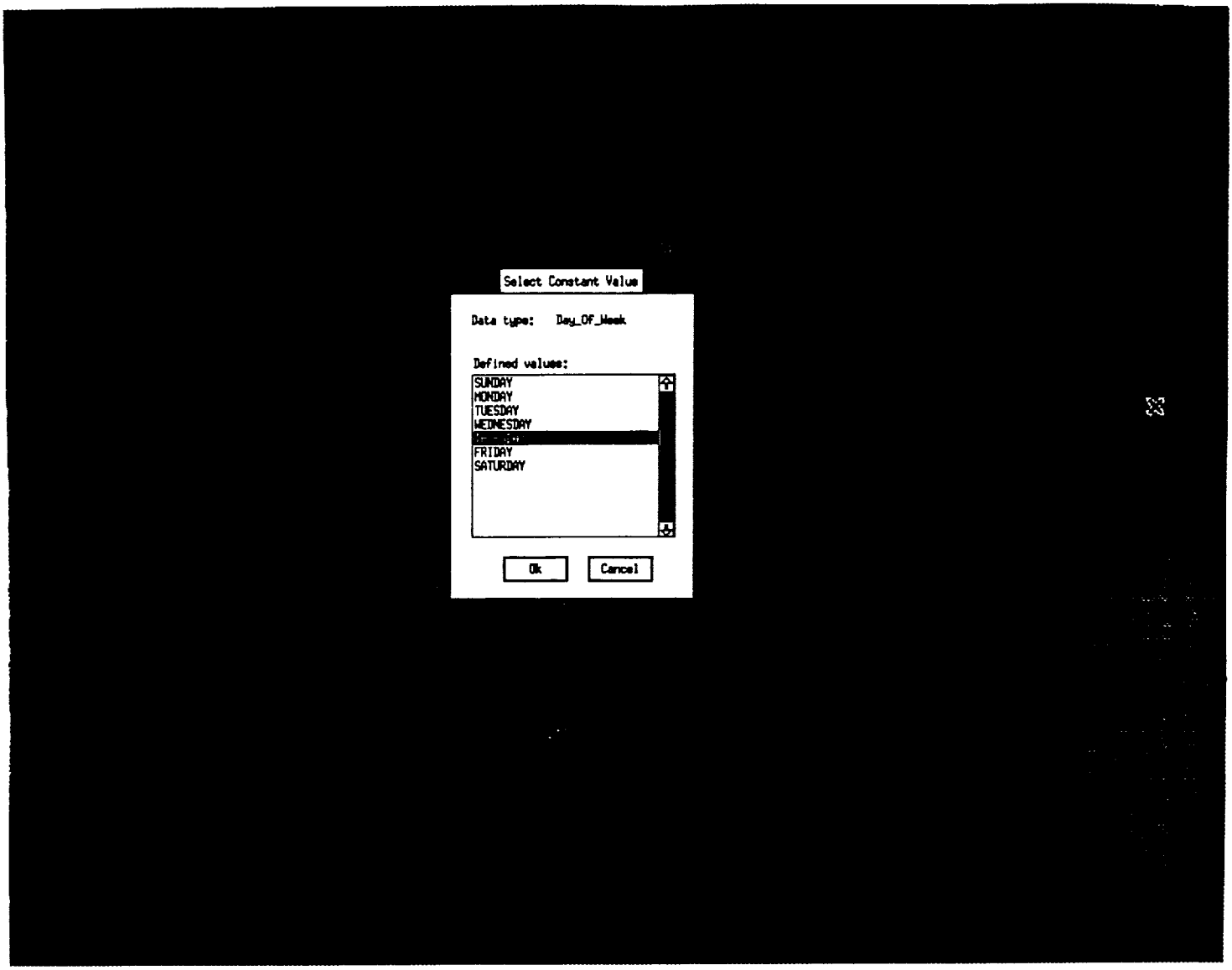

Select Constant Value
Data type: Day_Of_Week
Defined values:
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
FRIDAY
SATURDAY
Ok
Cancel

Figure 5-3: Select Constant Value Panel

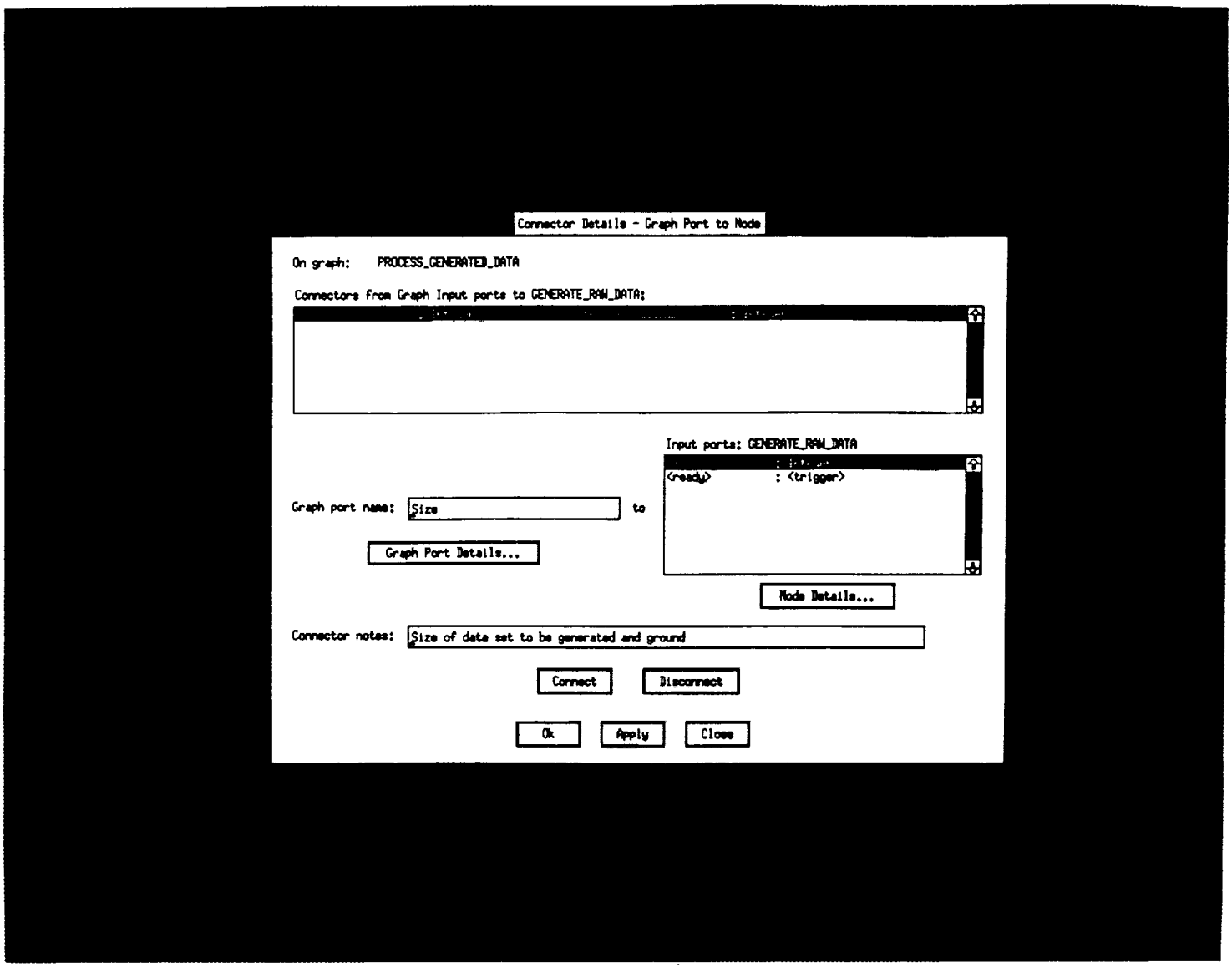
- **Disconnect.** In addition to the actions described in Section 5.2, selecting Disconnect also clears the Constant Value Field.

5.4 Graph Input Ports to Node Connector Details Panel

The Graph Input Ports to Node Connector Details Panel is used to connect graph input ports to a node's input ports. Instead of a Source Ports textlist, it has a text field for entering a name for a graph input port. This text field behaves like the Constant Value Field on the Constants to Node Connector Details Panel. The panel has a Graph Port Details... button instead of a Node Details... button. When the Graph Port Details... button is selected, a Graph Port Details Panel is popped up for the graph's input ports. The Graph Port Details Panel is discussed in Section 5.7. The Graph Input Ports to Node Connector Details Panel is shown in Figure 5-4.

The Node to Graph Output Ports Connector Details Panel is used to connect a node's ports to graph output ports. Its behaves analogously to the Graph Input Ports to Node

Figure 5-4


Connector Details - Graph Port to Node
On graph: PROCESS_GENERATED_DATA
Connectors from Graph Input ports to GENERATE_RAW_DATA:
Input ports: GENERATE_RAW_DATA
<ready> : <trigger>
Graph port name: Size
to
Graph Port Details...
Node Details...
Connector notes: Size of data set to be generated and ground
Connect
Disconnect
Ok
Apply
Close

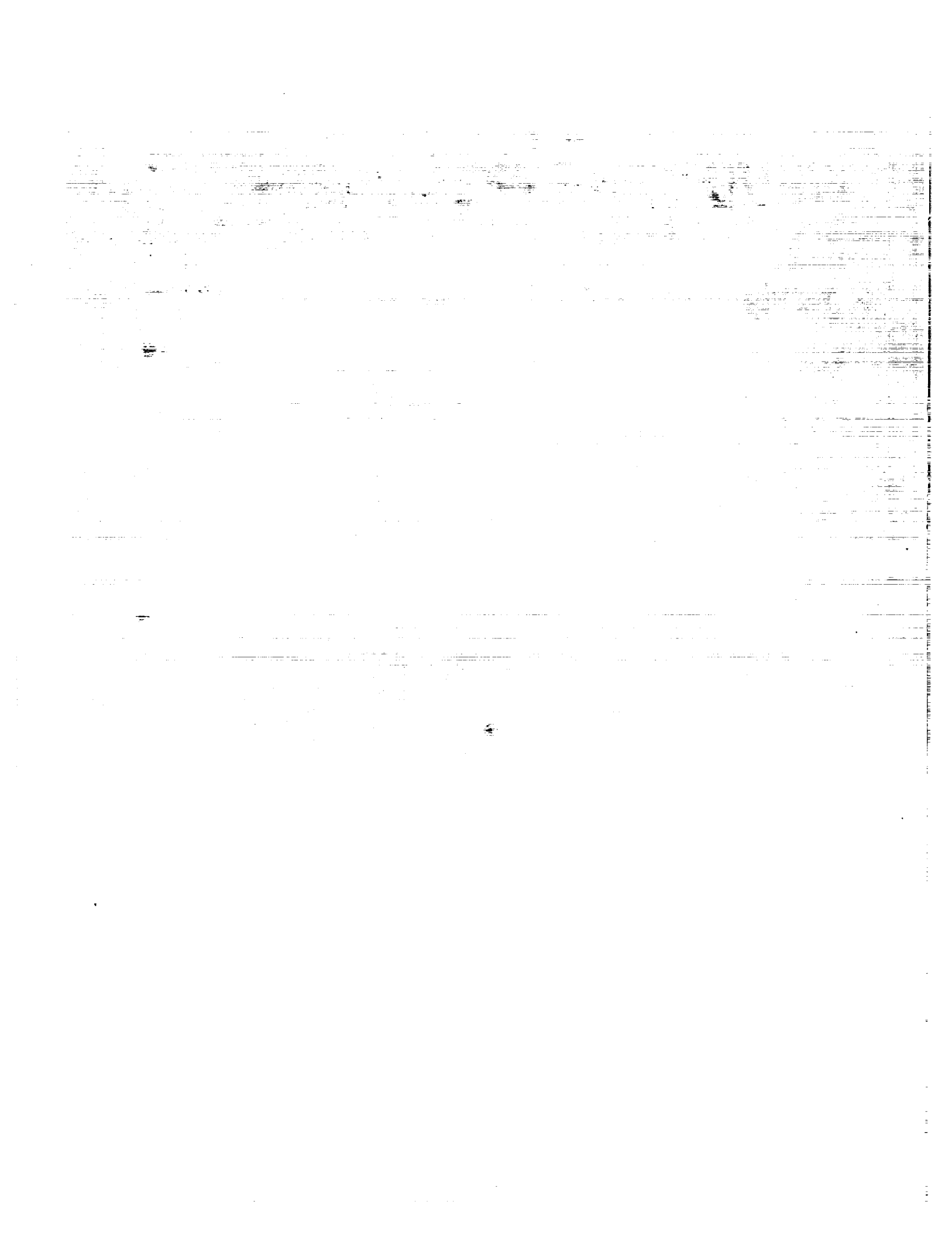


Figure 5-4: Graph Input Ports to Node Connector Details Panel

Connector Details Panel. The Node to Graph Output Port Connector Details Panel is shown in Figure 5-5.

When defining a connection to a graph port that is already connected to another port, the data types must be compatible. If this is not the case, a warning panel is popped up and the connection definition is not made.

5.5 The Node Details Panel

The Node Details Panel is used to view information about a node on a graph. From this panel the user may also change the name of the node or pop up a Notes Panel where notes about the node's use may be browsed or edited. The Node Details Panel is shown in Figure 5-6. The contents of the fields on the panel and responses to events are described below:

- **Name.** The Name Field is a text field which contains the node's name. Initially a node's name is system-generated. This field is read and the node's name is changed when the user terminates input to the field.

- **Type.** The Type Field identifies the type of of the node. Examples of values displayed in this field are "Procedure," "Function," "Subgraph," and "If." The text in this field is read only.

- **Input Ports and Output Ports.** The Input Ports and Output Ports Fields are textlist fields which list the input ports and output ports for the node and the connections they are associated with. Items in these fields give the port name, data type, and a connection status string.

  The current selection of either textlist is read when the Connector Details... Button immediately below the textlist is selected.

- **Connector Details...** Below both the Input Ports and Output Ports textlist fields is a Connector Details... button. When this button is selected, a Connector Details Panel is popped up for the connector group that corresponds to the current selection in the textlist. The Connector Details Panels are discussed in Sections 5.2, 5.3, and 5.4.

- **Component Details...** When the Component Details... Button is selected, a Component Details Panel is popped up for the subprogram corresponding to the node. The Component Details Panel is discussed in section 5.6.

- **Notes...** When the Notes... Button is selected, a Node Notes Panel is popped up. This panel is shown in Figure 5-7.

Figure 5-5

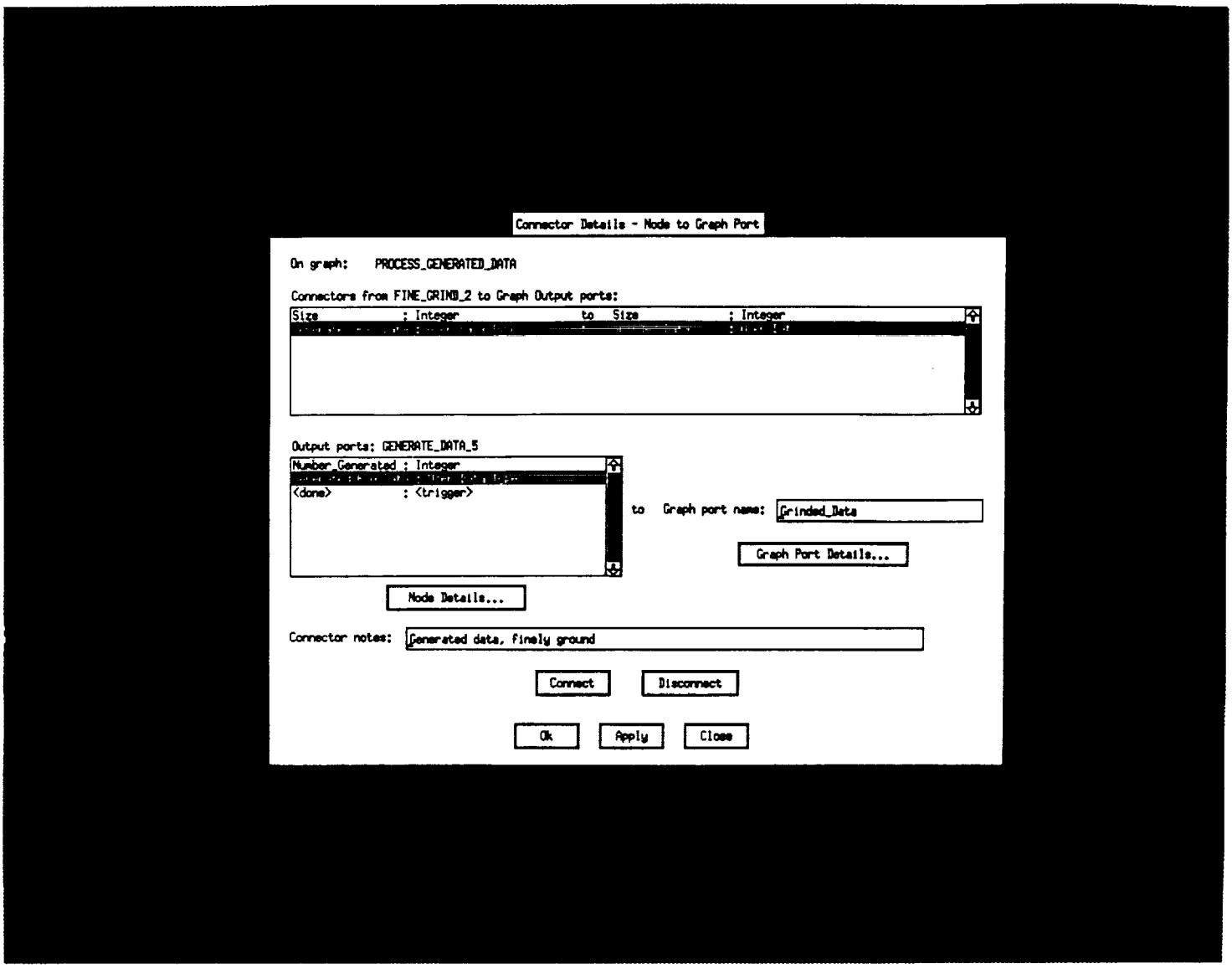

Connector Details - Node to Graph Port
On graph: PROCESS_GENERATED_DATA
Connectors from FINE_GRIND_2 to Graph Output ports:
Size : Integer to Size : Integer
Output ports: GENERATE_DATA_5
Number_Generated : Integer
<done> : <trigger>
to Graph port name: Grinded_Data
Graph Port Details...
Node Details...
Connector notes: Generated data, finely ground
Connect
Disconnect
Ok
Apply
Close

Figure 5-5: Node to Graph Output Ports Connector Details Panel

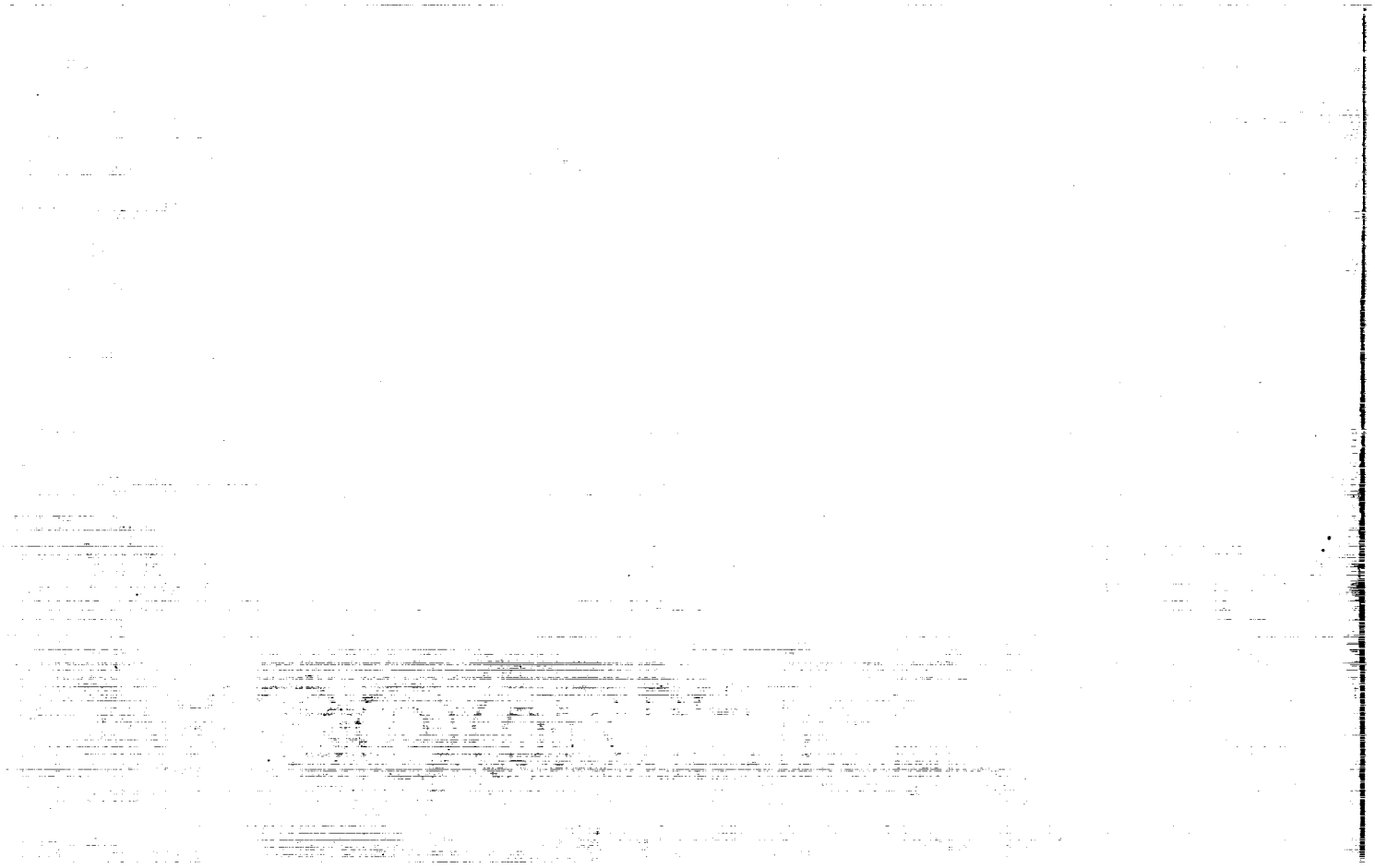


Figure 5-6

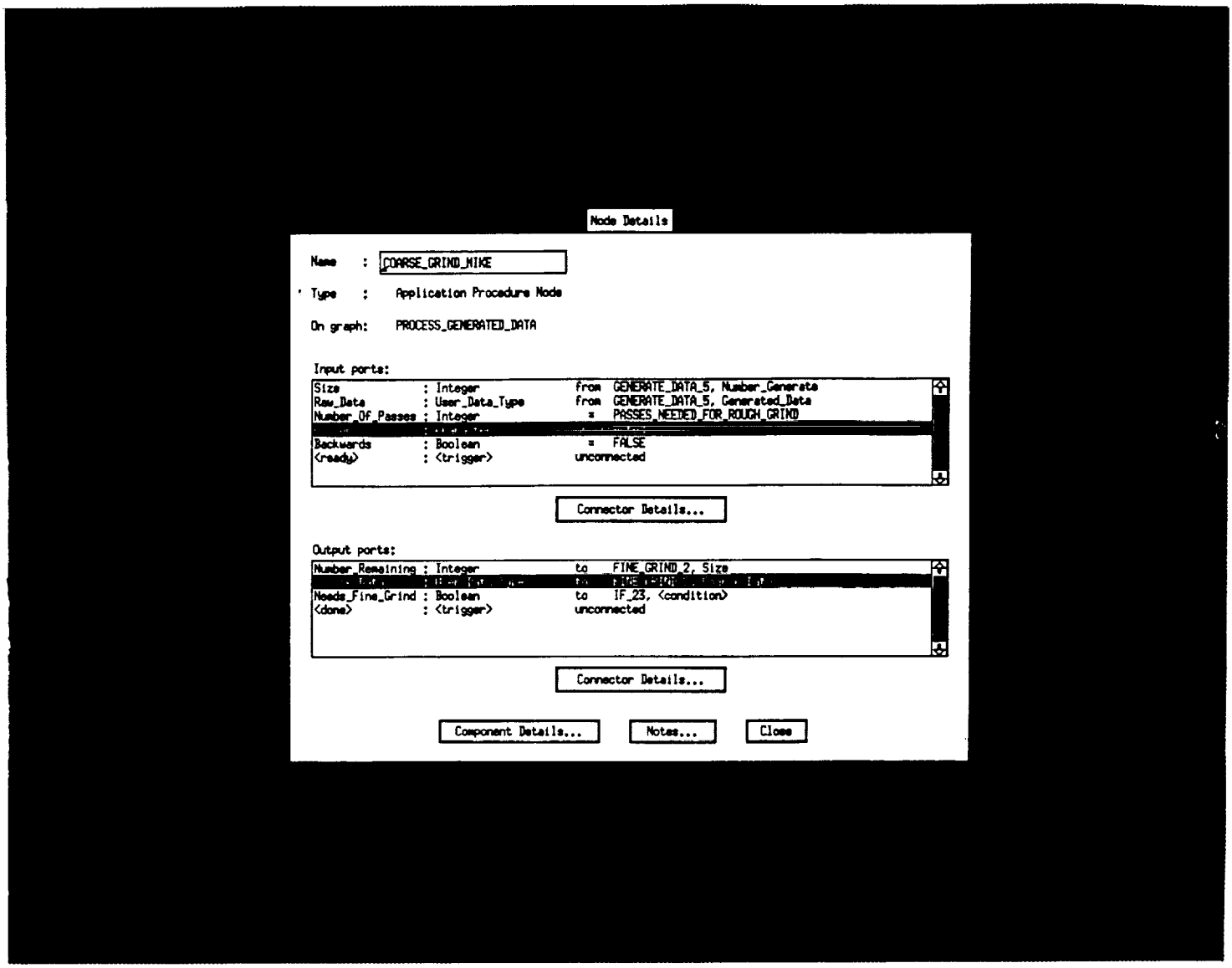

Node Details
Name : COARSE_GRIND_MIKE
Type : Application Procedure Node
On graph: PROCESS_GENERATED_DATA
Input ports:
Size : Integer from GENERATE_DATA_5, Number_Generate
Raw_Data : User_Data_Type from GENERATE_DATA_5, Generated_Data
Number_Of_Passes : Integer = PASSES_NEEDED_FOR_ROUGH_GRIND
Backwards : Boolean = FALSE
<ready> : <trigger> unconnected
Connector Details...
Output ports:
Number_Remaining : Integer to FINE_GRIND_2, Size
Needs_Fine_Grind : Boolean to IF_23, <condition>
<done> : <trigger> unconnected
Connector Details...
Component Details...
Notes...
Close

Figure 5-6: Node Details Panel

The Node Notes Panel contains fields that identify the node name and type, a field with Component comments, a field with user-editable notes about the node, an Ok button, and a Cancel button. When the Ok button is selected, changes to the node's notes are made permanent and the panel is dismissed. The Cancel button dismisses the panel without making any changes.

- **Close.** When the Close Button is selected, the Node Details Panel is dismissed.

5.6 Component Details Panel

The Component Details Panel is used to display information about a program component (procedure, function, or subgraph) which can be used as a node on another graph. The Component Details Panel is shown in Figure 5-8. The following describes the fields on this panel:

- **Name** and **Type.** The Name and Type Fields identify the component name and type. Component types are either Procedure, Function, or Graph. The text in these fields is read only.

- **Input Ports and Output Ports.** The Input Ports and Output Ports Fields list information about the component's input and output ports. Both fields are populated with lines that show the port name, data type, and port comments. For procedure or function components, the port comments are supplied by a Knowledge Engineer. For Graph components, the port comments are the same as the notes for the connector to the graph port. The text in these fields is read only.

- **Implementation** fields. The details about a component's implementation are listed in four implementation fields. The text in all these fields is read only. The fields are:

    - **Type.** This field contains a string indicating whether implementation type is Inline, Procedure, or Package.

    - **Package Name.** If the Implementation Type is Package, then this field contains the package name.

    - **Spec Filename.** If the Implementation Type is Package, then the filename for the Ada package spec is displayed in this field.

    - **Body Filename.** If the Implementation Type is Package or Procedure, then the filename for the Ada package or procedure body is displayed in this field.

Figure 5-7

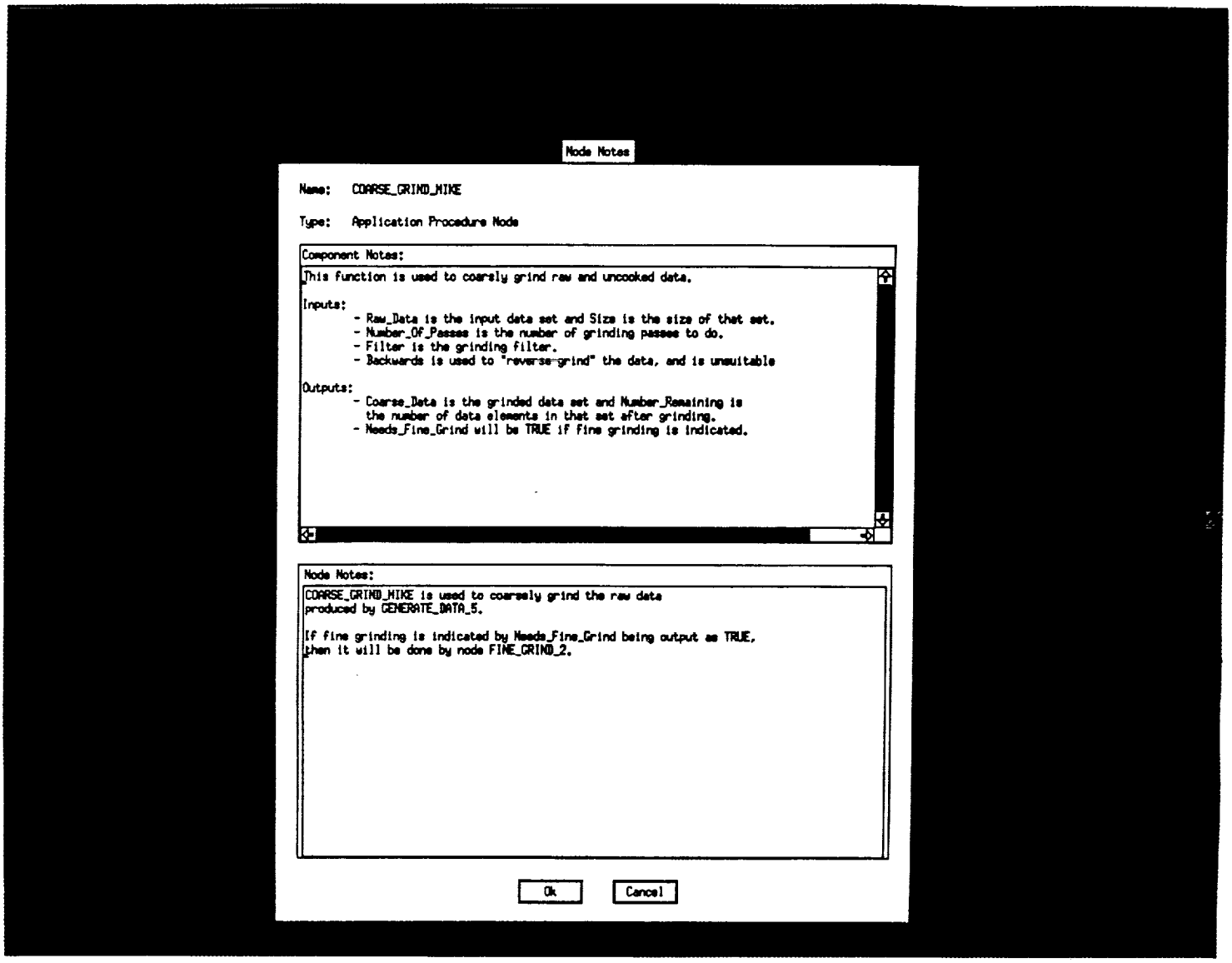

Node Notes
Name: COARSE_GRIND_MIKE
Type: Application Procedure Node
Component Notes:
This function is used to coarsly grind raw and uncooked data.
Inputs:
- Raw_Data is the input data set and Size is the size of that set.
- Number_Of_Passes is the number of grinding passes to do.
- Filter is the grinding filter.
- Backwards is used to "reverse-grind" the data, and is unsuitable
Outputs:
- Coarse_Data is the grinded data set and Number_Remaining is the number of data elements in that set after grinding.
- Needs_Fine_Grind will be TRUE if fine grinding is indicated.
Node Notes:
COARSE_GRIND_MIKE is used to coarsely grind the raw data produced by GENERATE_DATA_5.
If fine grinding is indicated by Needs_Fine_Grind being output as TRUE, then it will be done by node FINE_GRIND_2.
Ok
Cancel

Figure 5-7: Node Notes Panel

Figure 5-8

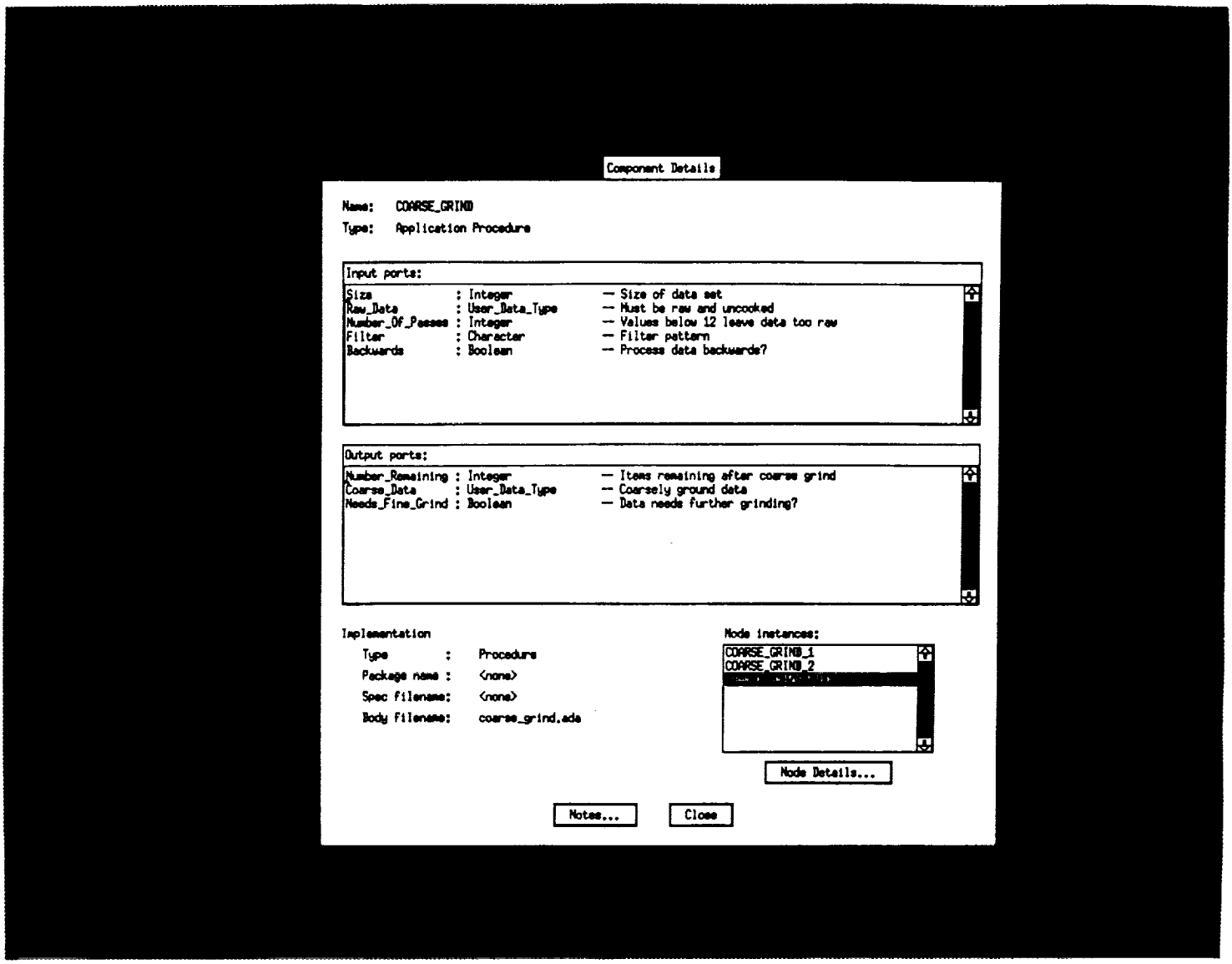

Component Details
Name: COARSE_GRIND
Type: Application Procedure
Input ports:
Size : Integer — Size of data set
Raw_Data : User_Data_Type — Must be raw and uncooked
Number_Of_Passes : Integer — Values below 12 leave data too raw
Filter : Character — Filter pattern
Backwards : Boolean — Process data backwards?
Output ports:
Number_Remaining : Integer — Items remaining after coarse grind
Coarse_Data : User_Data_Type — Coarsely ground data
Needs_Fine_Grind : Boolean — Data needs further grinding?
Implementation
Type : Procedure
Package name : <none>
Spec filename: <none>
Body filename: coarse_grind.ada
Node instances:
COARSE_GRIND_1
COARSE_GRIND_2
Node Details...
Notes...
Close

Figure 5-8: Component Details Panel

For Procedure or Function Components, the implementation information is contained in objects created by the Knowledge Engineer. For Graph Components, the implementation information is entered by the user.

- **Node Instances.** The Node Instances Field is a textlist field which contains the names of all nodes which use this program component. The current selection from this field read when the Node Details... Button is selected.

- **Node Details...** When the Node Details... button is selected, a Node Details Panel is popped up with information about the node corresponding to the currently selected item in the Node Instances textlist. The Node Details Panel is discussed in Section 5.5

- **Notes...** When the Notes... Button is selected, a Component Notes Panel is popped up. This panel is shown in Figure 5-9.

  The Components Notes Panel contains static text fields displaying the component name and type, a non-editable field of comments about this component, and a Close button. Selecting the Close Button dismisses the panel.

- **Close.** When the Close Button is selected, the Component Details Panel is dismissed.

5.7 Graph Port Details Panel

The Graph Port Details Panel is used to browse the set of input or output ports for a graph. This panel is popped up after the user has selected either the icon representing the current graph's input ports or the icon representing the current graph's output ports and then selected the Object Details... menu item on the Edit Menu. This panel is shown in Figure 5-10.

The fields on the Graph Port Details Panel are as follows:

- **Name.** The Name Field displays the graph name. The text in this field is read only.

- **Graph Ports.** The Graph Ports Field is a textlist field which lists information about the graph ports being examined. The title of the textlist identifies whether the panel is displaying the graph's input ports or its output ports. An item in the Graph Ports textlist gives the name of graph port, its data type, and its connection status.

  The current selection from this field is read when the Connector Details... button is selected.

Figure 5-9

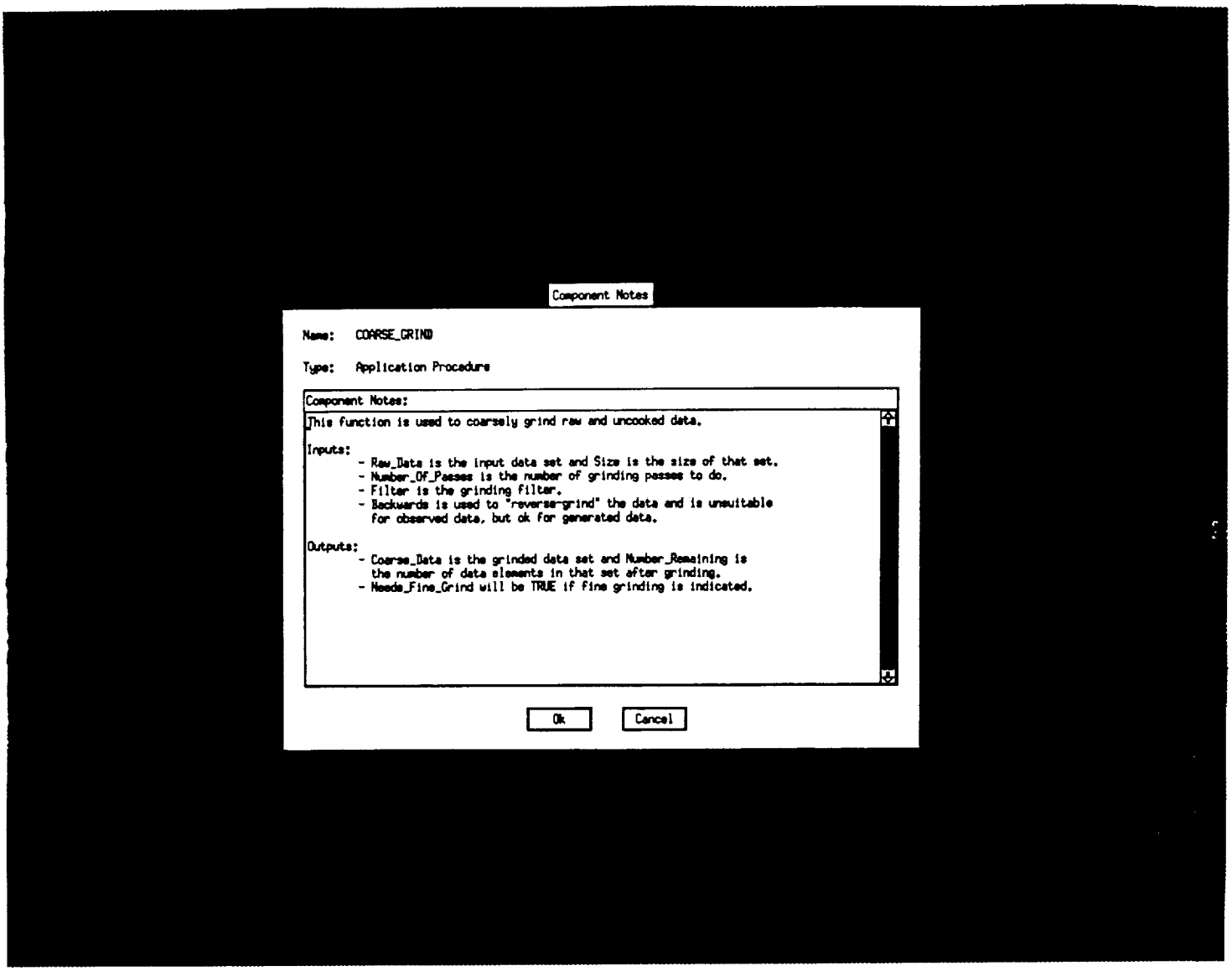

Component Notes
Name: COARSE_GRIND
Type: Application Procedure
Component Notes:
This function is used to coarsely grind raw and uncooked data.
Inputs:
- Raw_Data is the input data set and Size is the size of that set.
- Number_Of_Passes is the number of grinding passes to do.
- Filter is the grinding filter.
- Backwards is used to "reverse-grind" the data and is unsuitable for observed data, but ok for generated data.
Outputs:
- Coarse_Data is the grinded data set and Number_Remaining is the number of data elements in that set after grinding.
- Needs_Fine_Grind will be TRUE if fine grinding is indicated.
Ok
Cancel

Figure 5-9: Component Notes Panel

Figure 5-10

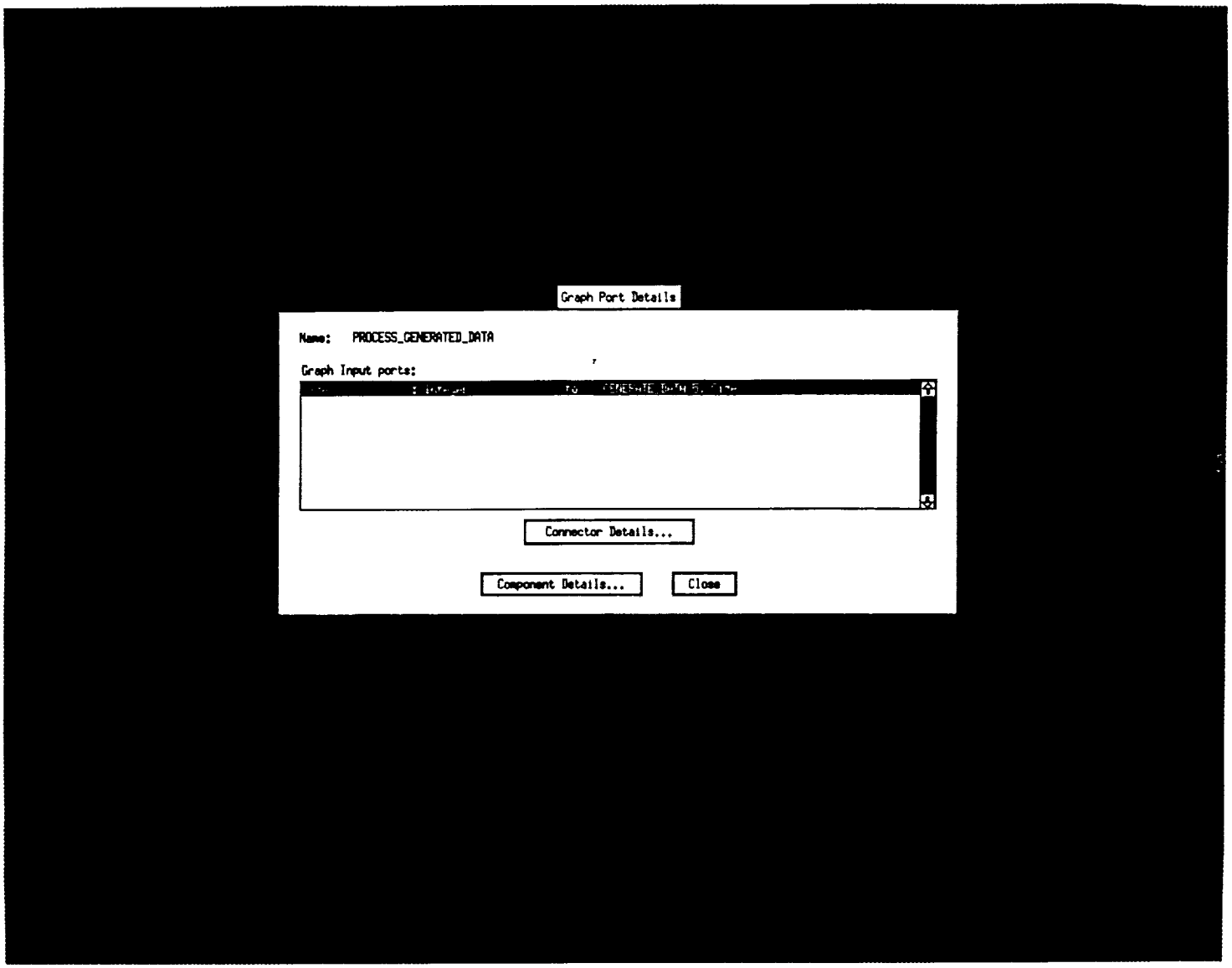
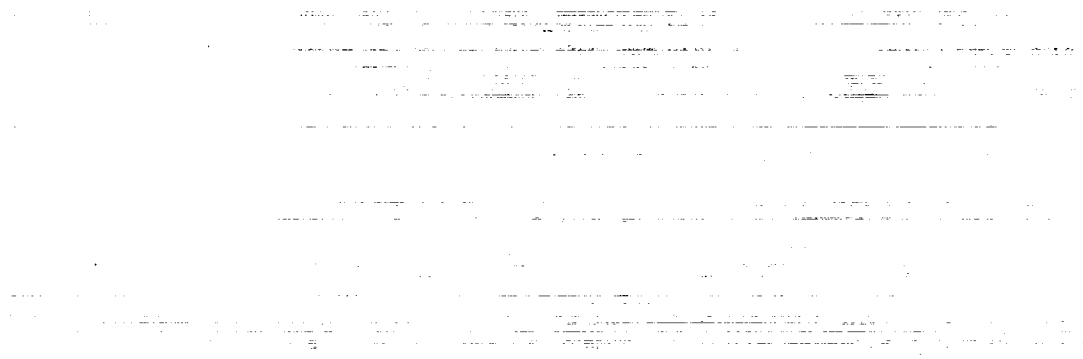

Graph Port Details
Name: PROCESS_GENERATED_DATA
Graph Input ports:
Connector Details...
Component Details...
Close

Figure 5-10: Graph Ports Details Panel



- **Connector Details...** When the Connector Details... Button is selected, a Connector Details Panel is popped up and displays information about the connector group corresponding to the currently selected item on the Graph Ports textlist. The Connector Details Panels are discussed in Section 5.2, 5.3, and 5.4.

- **Component Details...** When the Component Details... Button is selected, a Component Details Panel for the graph to which the graph ports belong is displayed. The Component Details Panel is discussed in section 5.6.

- **Close.** When the Close Button is selected, the Graph Port Details Panel is dismissed.

6. Graph Validation and Code Generation

[To be supplied]

I. ART-IM Schema Representation of ESL Objects

```
;;; Procedure and functions schemas correspond to ADA (or C)
;;; procedures and functions
;;; They may be grouped by the knowledge engineer into classes so that
;;; the can be readily browsed from the ACCESS tools panel.
;;; For example one might have a taxonomy of the following form:
;;;
;;;  hol-subprogram
;;;     mathematical-subprogram
;;;             trigonometric-function
;;;                  sin
;;;                  cos
;;;                  tan
;;;             inverse-trigonometric-function
;;;                  asin
;;;                  acos
;;;                  atan
;;;     list-manipulation-subprograms
;;;
;;;  etc.

;;;  The schemas at the lowest levels are instances (??????)

;; a program-component schema is the parent of procedure and graph
;; schemas
;;; SLOT Definitions

(defschema corresponds-to-nodes
        (instance-of slot)
        (cardinality MULTIPLE))

(defschema has-input-ports
        (instance-of slot)
        (cardinality MULTIPLE))

(defschema has-output-ports
        (instance-of slot)
        (cardinality MULTIPLE))

(defschema has-nodes
        (instance-of slot)
        (cardinality MULTIPLE))

(defschema has-connector-groups
        (instance-of slot)
        (cardinality MULTIPLE))

(defschema has-connectors
        (instance-of slot)
        (cardinality MULTIPLE))

;;;  SUBPROGRAM OBJECTS
```

```
(defschema subprogram
        (name)                  ;name of this component
        (notes)                 ;documentation for this procedure or
                                ;graph
        (has-implementation)    ;pointer to implementation schema
        (corresponds-to-nodes)  ;nodes which use this program component
        (has-input-ports)
        (has-output-ports)
)

(defschema primitive-subprogram
        (is-a subprogram)
        (subprogram-type)       ;for Ada value is procedure or function
;for documentation
;       (has-input-ports)       ;values are procedure-port schemas
;       (has-output-ports)      ;values are procedure-port schemas
        (has-visibility-requirements)
)

(defschema graph
        (is-a subprogram)
        (has-nodes)             ;multi-valued slot, values are nodes schemas
        (has-connector-groups)  ;multi-valued slot, values are
                                ;connector-group schemas
;       (has-input-ports)       ;values are graph-port schemas
;       (has-output-ports)      ;values are graph-port schemas

)

;;; NODE OBJECTS

;; a node is a box on a graph - so each node schema is "on" a particular
;; graph

(defschema node
        (on-graph)              ;value is a graph schema
        (name)                  ;user-supplied name for node
        (notes)                 ;value is documentation string
        (has-input-ports)
        (has-output-ports)
)

(defschema subprogram-node
        (is-a node)
        (uses-subprogram)
)

;; a node on a graph which is itself a graph

(defschema subgraph-node
        (is-a subprogram-node)
;       (uses-subprogram)       ;value is a graph schema - inverse
                                ;to corresponds-to-node
)
```

```
(defschema primitive-subprogram-node
        (is-a node)
;       (uses-subprogram)       ;value is primitive-subprogram schema -
                                ;inverse to corresponds-to-node
)

(defschema control-node
        (is-a node)
)

(defschema if-node
        (is-a control-node)
;       (has-input-ports)       ;value is single port whose data type
                                ;is boolean
;       (has-output-ports)      ;has two output ports of trigger type

)

(defschema select-node
        (is-a control-node)
;       (has-input-ports)       ;value is single port whose data type
                                ;is of enumerated type
;       (has-output-ports)      ;has multiple output ports, each named
                                ;to correspond with one of the
                                ;allowable values for the input port
                                ;data type
)

(defschema iterator-node
        (is-a control-node)
;       (has-input-ports)       ;value is single port whose data type
                                ;is boolean
;       (has-output-ports)      ;has two output ports of trigger type
)

(defschema replicator-node
        (is-a node)
;       (has-input-ports)       ;value is single port of any data type
;       (has-output-ports)      ;has multiple output ports with same
                                ;data type as the input port

)

(defschema merge-node
        (is-a node)
;       (has-input-ports)       ;has two or more input ports with same
                                ;data type
;       (has-output-ports)      ;has single output port of same data
                                ;type as the input port

)

;;; PORT SCHEMAS

(defschema port
        (name)
        (direction)             ;value is INPUT or OUTPUT
        (port-data-type)        ;value is string which corresponds to
                                ;data-type schema or "trigger"
)
```

```
(defschema graph-port
        (is-a port)
        (on-graph)
)

;; a primitive-subprogram-port schema represents the parameter to a
;procedure or function

(defschema primitive-subprogram-port
        (is-a port)
        (belongs-to-subprogram) ;value is primitive-subprogram schema
        (position)              ;position of this parameter  in
                                ;calling sequence for procedure or function
        (parameter-type)        ;IN, OUT, IN-OUT, or RETURN-VALUE
)

(defschema node-port    ;used for instances of predefined nodes
        (is-a port)
        (on-node)
)

;;; DATA-TYPE SCHEMAS

;; naming convention is name of data type with "-DATA-TYPE-SPEC"
;; appended  - e.g., INTEGER-DATA-TYPE-SPEC
;;
(defschema data-type
        (name-of-data-type)     ;value is string
        (defined-values)        ;values are strings
        (test-function)         ;boolean function which accepts a
                                ;string token
                                ;as input and determines if it is an
                                ;allowable identifier for this data type
)

(defschema constant
        (has-data-type)         ;value is data-types schema
        (value)                 ;value is string such that the
                                ;function referenced from the
                                ;test-function slot of the data-types
                                ;schema, when applied to this string,
                                ;returns T
)

;;; CONNECTOR and CONNECTOR GROUP SCHEMAS

;;; HAVE PUT IN CONNECTOR-GROUP SCHEMA so that logical organization
;;; corresponds with graphical representation  - one could do away
;;; with the connector-group schema
```

```
(defschema connector-group
        (on-graph)              ;inverse of has-connector-groups
        (source-node)           ;value is node schema which is on the
                                ;same graph as this connector or
                                ;CONSTANT-VALUE or GRAPH-INPUTS
        (destination-node)      ;value is node schema which is on the
                                ;same graph as this connector or
                                ;CONSTANT-VALUE  or GRAPH-OUTPUTS
        (has-connectors)        ;values are connector schemas
)

(defschema connector
        (notes)                 ;value is documentation string
        (belongs-to-connector-group)    ;value is connector-group schema
;;
;; the type of value for the source-port  slot depends on
;; the value of the source-node slot in the connector-group
;; schema as follows:
;;
;; if the value of the source-node slot is
;;
;;    a) a node schema , then the value of the
;;       source-port slot must be one of the values of the
;;       has-output-ports slot of the node schema
;;
;;    b) CONSTANT-VALUE, then the value of the source-port slot
;;       must be a constant schema
;;
;;    c) GRAPH-INPUTS then the value of the source-port slot
;;       be a graph-port schema whose direction is INPUT
;;
        (source-port)
;;
;; analogous constraints apply to the value of the
;; destination-node
;;
        (destination-port)
)

;;; IMPLEMENTATION SCHEMAS

(defschema implementation)

(defschema inline-implementation
        (is-a implementation)
        (code-template)
)

(defschema separately-compiled-procedure
        (is-a implementation)
        (source-file-name)
        (object-file-name)
        (library-file-name)
)
```

```
(defschema package-implementation
        (is-a implementation)
        (name-of-package)
        (has-procedures)
        (package-spec-file-name)
        (package-body-file-name)
        (package-object-spec-file-name)
        (package-body-object-file-name)
)
```